M4GB: An Efficient Gröbner-Basis Algorithm

Rusydi H. Makarim
Cryptology Group, Centrum Wiskunde en Informatica
Amsterdam, Netherlands
Mathematical Institute, University Leiden
Leiden, Netherlands
makarim@cwi.nl

Marc Stevens
Cryptology Group, Centrum Wiskunde en Informatica
Amsterdam, Netherlands
marc.stevens@cwi.nl

ABSTRACT

This paper introduces a new efficient algorithm for computing Gröbner-bases named M4GB. Like Faugère's algorithm F4 it is an extension of Buchberger's algorithm that describes: how to store already computed (tail-)reduced multiples of basis polynomials to prevent redundant work in the reduction step; and how to exploit efficient linear algebra for the reduction step. In comparison to F4 it removes further redundant work in the processing of reducible monomials. Furthermore, instead of translating the reduction of many critical pairs into the row reduction of some large matrix, our algorithm is described more natively and is efficient while processing critical pairs one by one. This feature implies that typically M4GB has to process fewer critical pairs than F4, and reduces the time and data complexity 'staircase' related to the increasing degree of regularity for a sequence of problems one observes for F4.

To achieve high efficiency, M4GB has been designed specifically to operate only on tail-reduced polynomials, i.e., polynomials of which all terms except the leading term are non-reducible. This allows it to perform full-reduction directly in the computation of a term polynomial multiplication, where all computations are done over coefficient vectors over the non-reducible monomials.

We have implemented a version of our new algorithm tailored for dense overdefined polynomial systems as a proof of concept and made our source code publicly available. We have made a comparison of our implementation against the implementations of FGBlib, Magma and OpenF4 on various dense Fukuoka MQ challenge problems that we were able to compute in reasonable time and memory. We observed that M4GB uses the least total CPU time *and* the least memory of all these implementations for those MQ problems, often by a significant factor.

In the Fukuoka MQ challenges, the starting challenges of Type V and Type VI have 16 equations which was chosen based on an extrapolated computational runtime of more than a month using Magma. M4GB allowed us to set new records for these Fukuoka MQ challenges breaking Type V ($\mathbb{F}_{2^8}$) up to 18 equations and Type VI ($\mathbb{F}_{31}$) up to 19 equations, each can be computed within up to 11 days on our dual Xeon system.

ISSAC '17, July 25-28, 2017, Kaiserslautern, Germany

ACM ISBN 978-1-4503-5064-8/17/07…$15.00
https://doi.org/10.1145/3087604.3087638

KEYWORDS

Gröbner basis algorithm; multivariate polynomial systems; quantum-safe public key crypto

ACM Reference format:
Rusydi H. Makarim and Marc Stevens. 2017. M4GB: An Efficient Gröbner-Basis Algorithm. In *Proceedings of ISSAC '17, Kaiserslautern, Germany, July 25-28, 2017,* 8 pages.
https://doi.org/10.1145/3087604.3087638

1 INTRODUCTION

For several decades, the design of trapdoor functions in public-key cryptography mostly relied on hardness of integer factorization and discrete logarithms in finite fields and algebraic curves. However, as Shor's algorithm [14] yields a polynomial-time quantum algorithm for integer factorization and discrete logarithm, several alternative computational hard problems have been proposed to design new public-key and digital signature schemes that are more resistant against quantum attacks. In this paper we focus on one of these, namely the multivariate polynomial (MP) problem.

Problem 1 (MP-problem). *Let $n, m \in \mathbb{Z}_{>0}$ and let $\mathbb{F}$ be a field. Given m polynomials $f_1, \ldots, f_m \in \mathbb{F}[x_1, \ldots, x_n]$, find $(s_1, \ldots, s_n) \in \mathbb{F}^n$ such that $f_i(s_1, \ldots, s_n) = 0$ for all $i = 1, \ldots, m$.*

The special case when all f_i are quadratic is called the multivariate quadratic (MQ) problem, and both the MP and the MQ problem are well known to be NP-hard.

A generic method to solve these polynomial equations is by computing a Gröbner basis of the ideal generated by $f_1, f_2, \ldots, f_m$. The notion of Gröbner bases and the classical algorithm to compute them were introduced by Bruno Buchberger in 1965 [3]. The F_4 algorithm from Faugère [8] is significantly more efficient than Buchberger's as it converts the classical reduction step for many polynomials to the row reduction of a single sparse matrix, which can be performed by optimized linear algebra packages. Another practical algorithm to solve MQ-problems is the XL algorithm [4, 6] that, similar to F4, uses linear algebra on large matrices involving polynomials $m_j \cdot f_i$ with all monomials m_j up to a given degree.

Since the MQ-problem is one of the candidate problems to build quantum-safe public-key cryptographic primitives, it is essential to analyze its practical difficulty as a hard computational problem. Public challenges for several other computational hard problems have been publicly announced such as the RSA challenge [13], the Lattice challenge [15], and the ECC challenge [7]. With a similar goal in mind, Yasuda et al. [16] started public challenges for MQ-problems[1] in April 2015. They pose random MQ challenges of six

[1] https://www.mqchallenge.org/

types, namely for two modes combined with three finite fields, and for increasing security parameter (i.e., number of variables). Types I, II, and III are Encryption-type problems, where the number of equations is twice the number of variables, with randomly-chosen coefficients in $\mathbb{F}_2, \mathbb{F}_{256}$ and $\mathbb{F}_{31}$, respectively. Types IV, V, and VI are Digital-Signature-type problems, where the number of equations is two-thirds the number of variables, with the same respective base fields $\mathbb{F}_2, \mathbb{F}_{256}, \mathbb{F}_{31}$. For each problem type, the MQ-Challenge organizers chose the starting number of variables as the smallest number that expectedly takes at least a month to solve using Magma version 2.19-9 running on four 6-cores Intel Xeon CPU E5-4617 2.9GHz with 1TB of RAM.

Our contribution. In this paper we introduce the M4GB algorithm as a special version of Buchberger's algorithm. M4GB was designed with the specific aim to perform all computations on tail-reduced polynomials. To that end, it stores not only the monic intermediate basis polynomials, but also monomial multiples thereof in one large set M of tail-reduced polynomials. The intention is to store these polynomials as pairs consisting of a Leading Monomial and a non-reducible tail, in order to facilitate efficient linear algebra over these tails. More specifically, using a global ordered set of necessary non-reducible monomials one can ensure every Tail can be stored as a dense sequence of field coefficients. Thence one can view this set M as one large matrix, where columns are labeled by non-reducible monomials, each row contains the Tail of a polynomial and is labeled by the Leading Monomial of that polynomial. This has the benefit that for the multiplication of a non-reducible Tail g with a monomial u one can directly compute the full-reduced outcome. Namely, for $c_i m_i \in g$ either the term $c_i m_i u$ is a non-reducible term and thus can be directly added to the outcome, or is reducible and then instead $c_i h$ is subtracted from the outcome, where h is the Tail belonging to the polynomial with Leading Monomial $m_i u$ retrieved from M.

We have implemented a proof-of-concept of M4GB in C++11 in a flexible framework. It features fast order-preserving encoding and decoding of monomial into integers, a simple small finite field implementation based on log/exp-tables, special postprocessing of new basis polynomials and multi-threading as discussed in Section 5. We have made our source code publicly available for the benefit of the community, as well as to verify our results.

Results from benchmarking our M4GB implementation against several existing implementation of Gröbner basis algorithms such as FGbLib[9], OpenF4[5], and MAGMA[2] using MQ-problems show a significant improvement in both total CPU time and total memory usage. Moreover, with M4GB we were able to solve several sizes of the signature-type MQ challenges for $\mathbb{F}_{256}$ and $\mathbb{F}_{31}$. Even using one desktop machine with an Intel i7-2600K CPU and 16 Gigabytes of RAM, we managed to solve the smallest challenge ($m = 16$ equations) for type V and VI within 9.3 hours and 1.2 hours, respectively. Furthermore, we were able to solve Type V challenges with $m = 17$ and $m = 18$ number of equations, as well as type VI challenges for $m = 17, 18, 19$. All these parameters are broken in far less than a month, the designed runtime for $m = 16$ from the MQ challenge organizers. A summary of these results is available in Table 3.

Outline. This paper is organized as follows. First, we introduce notations and preliminaries in Section 2. We present the high-level description of M4GB algorithm in Section 4, followed by further M4GB implementation details and features in Section 5. Section 5 also discusses the result from benchmarking our M4GB implementation against several existing implementation of Gröbner basis algorithm.

2 PRELIMINARIES

In this section, we recall some basic notions and results related to Gröbner bases. Even though the concepts and notions related to Gröbner bases work over any field, in this paper we restrict our discussion over a finite field $\mathbb{F}_q$.

Let $\mathcal{R} = \mathbb{F}_q[x_1, \ldots, x_n]$ be the polynomial ring over finite field $\mathbb{F}_q$ with n variables. We call $x_1^{a_1} \cdots x_n^{a_n}$ with $a_1, \ldots, a_n \in \mathbb{Z}_{\geq 0}$ a monomial and $a_1 + \ldots + a_n$ is its degree. Let

$$\mathcal{M} = \{x_1^{a_1} \cdots x_n^{a_n} : a_1, \ldots, a_n \in \mathbb{Z}_{\geq 0}\}$$

be the set of all monomials. A term is a product cu with $c \in \mathbb{F}_q, u \in \mathcal{M}$ and $\mathcal{T} = \{cu : c \in \mathbb{F}_q, u \in \mathcal{M}\}$ is the set of all terms. Given a non-zero polynomial $f = \sum_i c_i u_i \in \mathcal{R}$, with $c_i \in \mathbb{F}_q$ and $u_i \in \mathcal{M}$, we define $\mathsf{Term}(f) = \{c_i u_i : c_i \neq 0\}$ and $\mathsf{Mono}(f) = \{u_i : c_i \neq 0\}$ as the set of all terms, respectively monomials, of f.

Let $\prec$ be an admissible monomial ordering over $\mathcal{M}$, e.g., the degree reverse lexicographical order. The leading monomial $\mathsf{LM}(f) = \max_\prec \mathsf{Mono}(f) = u_i$ is the largest monomial u_i of f with non-zero coefficient c_i, then f's leading coefficient is $\mathsf{LC}(f) = c_i$ and leading term is $\mathsf{LT}(f) = \mathsf{LC}(f)\mathsf{LM}(f) = c_i u_i$. We call $\mathsf{Tail}(f) = f - \mathsf{LT}(f)$ the tail of f. The polynomial f is monic if $\mathsf{LC}(f) = 1$.

We extend the functions $\mathsf{LM}(f)$, $\mathsf{LT}(f)$ and $\mathsf{Tail}(f)$ to a set S of polynomials as follows: $\mathsf{Func}(S) = \{\mathsf{Func}(f) \mid f \in S\}$. Similarly we extend $\mathsf{Mono}(f)$ and $\mathsf{Term}(f)$ to a set S of polynomials by taking the union of their outputs: $\mathsf{Func}(S) = \bigcup_{f \in S} \mathsf{Func}(f)$.

Let $I \subseteq R$ be a polynomial ideal. Let $G = \{g_1, \ldots, g_t\} \subset \mathcal{R}$ be a finite set of polynomials. Then G is a *basis* for I if and only if $I = \langle G \rangle$ is generated by G. Furthermore, G is called a *Gröbner basis* of I if and only if for all $f \in I$ there exists $g \in G$ such that $\mathsf{LM}(g) \mid \mathsf{LM}(f)$.

We call a monomial u *reducible by* G if $\exists g \in G : \mathsf{LM}(g) \mid u$, i.e., there exists a $g \in G$ whose leading monomial divides u, and u is *non-reducible by* G if no such g exists. We assume a deterministic function $\mathsf{ReduceSel}(G, u) \to g$ that for a monomial u reducible by G outputs a $g \in G$ such that $\mathsf{LM}(g) \mid u$.

We call a polynomial $f \in \mathcal{R}$ *lead-reducible by* G if $\mathsf{LM}(f)$ is reducible by G. We call f *full-reducible by* G if there exists $u \in \mathsf{Mono}(f)$ that is reducible by G. Furthermore, we define that f is *tail-reducible by* G if $\mathsf{Tail}(f)$ is full-reducible by G.

To lead-reduce f by G we define:

$$\textsc{LeadReduce}(f, G) = \begin{cases} 0 & \text{if } f = 0; \\ f & \text{if } \mathsf{LM}(f) \text{ non-reducible by } G; \\ \textsc{LeadReduce}(f - \frac{\mathsf{LT}(f)}{\mathsf{LT}(h)} h, G) & \text{where } h \leftarrow \mathsf{ReduceSel}(G, \mathsf{LM}(f)). \end{cases}$$

It thus either returns f if it is not lead-reducible or calls itself on a polynomial with smaller leading monomial obtained by canceling the leading monomial of f with a multiple of some $g \in G$.

Algorithm 1: Algorithm Buchberger

Input: A finite subset F of $\mathcal{R}$
Output: A Gröbner basis G of $\langle F \rangle$.

```
1  G ← ∅; P ← ∅; // Basis G, set P of critical pairs (f, g)
2  for f ∈ F do
3      f ← FullReduce(f, G);
4      (G, P) ← Update(G, P, f);
5  while P ≠ ∅ do        // Process a Critical Pair until P is empty
6      (f, g) ← Select(P);
7      P ← P \ {(f, g)};
8      h ← FullReduce(Spoly(f, g), G);
9      if h ≠ 0 then
10         (G, P) ← Update(G, P, h);
11         G ← {TailReduce(g, G) : g ∈ G}; // Interreduce
12 return G;
```

To full-reduce f by G we define:

$$\textsc{FullReduce}(f,G) = \begin{cases} 0 & \text{if } f = 0; \\ \mathsf{LT}(f) + \textsc{FullReduce}(\mathsf{Tail}(f), G) & \text{if } \mathsf{LT}(f) \text{ non-reducible by } G; \\ \textsc{FullReduce}(f - \frac{\mathsf{LT}(f)}{\mathsf{LT}(h)} h, G) & h \leftarrow \mathsf{ReduceSel}(G, \mathsf{LM}(f)). \end{cases}$$

Finally, we define the tail-reduction of f by G as

$$\textsc{TailReduce}(f,G) = \mathsf{LT}(f) + \textsc{FullReduce}(\mathsf{Tail}(f), G).$$

Note that LeadReduce, FullReduce and TailReduce are deterministic as ReduceSel is. Also, they terminate as for all inputs f any recursion is called on a polynomial g with $\mathsf{LM}(g) \prec \mathsf{LM}(f)$ and this monotonic decreasing sequence of leading monomials is finite and has a smallest element, because $\prec$ is a well-ordering.

A basis G is called *minimal* if for all $g \in G$ the polynomial g is monic and g is not lead-reducible by $G \setminus \{g\}$. We say that a basis G is *tail-reduced* if every polynomial in G is monic and for all $g \in G$ the polynomial g is not tail-reducible by $G \setminus g$. Finally, G is called a *reduced* basis if it is both minimal and tail-reduced.

3 GRÖBNER BASIS ALGORITHMS

Buchberger [3] proved that a basis G for an Ideal is a Gröbner basis if and only if $\textsc{LeadReduce}(\mathsf{Spoly}(f,g), G) = 0$, for all $f, g \in G$, where $\mathsf{Spoly}(f,g)$ is the so-called *S-polynomial* of f and g:

$$\mathsf{Spoly}(f,g) = \frac{u}{\mathsf{LT}(f)} \cdot f - \frac{u}{\mathsf{LT}(g)} \cdot g, \quad u = \mathsf{LCM}(\mathsf{LM}(f), \mathsf{LM}(g))$$

He then proposed an algorithm to compute a Gröbner basis from a given basis G by repeatedly adding new non-zero polynomials to G that are found by lead-reducing $\mathsf{Spoly}(f,g)$ for all pairs $f, g \in G$ until this is not possible anymore. It maintains a list of critical pairs $P \in G \times G$ of pairs $f, g \in G$ to be tried, where some pairs can already be eliminated based on criteria that imply that the corresponding S-poly lead-reduces to 0. Buchberger proved this process always terminates and that the resulting G is a Gröbner basis.

In algorithm 1 we give a description of a well-known variant of Buchberger's algorithm that maintains a tail-reduced basis G. Selection of a critical pair $(f,g) \in P$ is given by a deterministic function $\textsc{Select}(P) \to (f,g)$. A common good choice is to always selects a critical pair (f,g) with smallest $\mathsf{LCM}(\mathsf{LM}(f), \mathsf{LM}(g))$. To update the basis and list of critical pairs, it uses the function

$$\textsc{Update}(G_{\text{old}}, P_{\text{old}}, f) \to (G_{\text{new}}, P_{\text{new}}).$$

It is recommended to use the Gebauer-Möller Installation [12] that maintains a minimal basis G. In this paper we assume that Update does not reduce basis elements itself and therefore operates strictly on the leading monomials of the basis elements, more specifically using the embedding of $\mathcal{M}$ in $\mathcal{R}$ we assume that:

$$(\mathsf{LM}(G_{\text{new}}), \mathsf{LM}(P_{\text{new}})) = \textsc{Update}(\mathsf{LM}(G_{\text{old}}), \mathsf{LM}(P_{\text{old}}), \mathsf{LM}(f)), \tag{1}$$

where $\mathsf{LM}(P) = \{(\mathsf{LM}(f), \mathsf{LM}(g)) : (f,g) \in P\}$ denotes the extension of LM to a set of pairs of polynomials.

An improvement by Faugère in his seminal paper on the F_4 algorithm [8] gave a new direction in the development of Gröbner bases algorithms. It performs polynomial reduction using efficient linear algebra, by translating the reduction of many S-polynomials into the row reduction of a coefficient matrix corresponding to the S-polynomials and necessary reductor polynomials. Also, it tracks full-reduced polynomials in these row-reduced matrices so it can reuse these in subsequent steps using a '`Simplify`'-function thereby removing a lot of redundant computations between iterations.

3.1 Cryptographic problems

Note that many cryptographic problems can be written as a system of equations $f_1 = 0, \ldots, f_m = 0$ for $f_1, \ldots, f_m \in \mathbb{F}_q[x_1, \ldots, x_n]$. If this has a unique solution then one can directly read the solution from a computed Gröbner basis for the ideal $\langle f_1, \ldots, f_m \rangle$. In more general settings, for a zero-dimensional ideal one normally computes a Gröbner basis w.r.t. an efficient monomial ordering to minimize computational cost, typically the degree-reverse lexicographic ordering. Then, using FGLM algorithm [10] which has polynomial complexity in the number of solutions, the Gröbner basis can be converted to another Gröbner basis w.r.t. lexicographic monomial ordering, allowing us to obtain the set of solutions.

4 M4GB ALGORITHM

We introduce our M4GB algorithm presented in algorithm 2 as an extension of Buchberger's algorithm, designed with the specific aim to perform all computations on tail-reduced polynomials. It performs the same essential steps as algorithm 1. However, M4GB has the following modifications compared to algorithm 1.

First, the tail-reduced basis G is replaced by (L, M), where $G \subset M$ and $L = \mathsf{LM}(G)$, that satisfy the M4GB-invariant:

Definition 4.1 (M4GB Invariant). A pair (L, M) with $L \subset \mathcal{M}$, $M \subset \mathcal{R}$ is said to satisfy the *M4GB-invariant* if and only if:

- $\forall f, g \in M : \mathsf{LM}(f) = \mathsf{LM}(g) \Leftrightarrow f = g$, i.e., every leading monomial occurs once;
- $L \subseteq \mathsf{LM}(M)$;
- Every $f \in M$ is lead-reducible by L, thus any monomial is reducible by M if and only if it is reducible by L.
- No $f \in M$ is tail-reducible by L;

In this section we often refer to the basis $G = \{g \in M : \mathsf{LM}(g) \in L\}$ implied by (L, M). The set M not only contains this tail-reduced

Algorithm 2: Algorithm M4GB

Input: A finite subset F of $\mathcal{R}$
Output: A Gröbner basis G of $\langle F \rangle$.

$L, M \leftarrow \emptyset, \emptyset$; // $G = \{g \in M : \mathsf{LM}(g) \in L\}$ *is tail-reduced basis*
$P \leftarrow \emptyset$; // *Set of critical pairs* (f_{LM}, g_{LM})
for $f \in F$ **do**
  $(M, f) \leftarrow$ MulFullReduce$(L, M, 1, f)$;
  $(L, M, P) \leftarrow$ UpdateReduce(L, M, P, f);
while $P \neq \emptyset$ **do** // *Process Critical Pair*
  $(f_{LM}, g_{LM}) \leftarrow$ Select(P);
  $P \leftarrow P \setminus \{(f_{LM}, g_{LM})\}$;
  $(f, g) \leftarrow (M[f_{LM}], M[g_{LM}])$;
  $u \leftarrow \mathsf{LCM}(f_{LM}, g_{LM})$;
  $(M, h_1) \leftarrow$ MulFullReduce$(L, M, u/\mathsf{LT}(f), \mathsf{Tail}(f))$;
  $(M, h_2) \leftarrow$ MulFullReduce$(L, M, u/\mathsf{LT}(g), \mathsf{Tail}(g))$;
  $h \leftarrow h_1 - h_2$; // *Full-reduction of* $\mathsf{Spoly}(f, g)$
  if $h \neq 0$ **then**
    $(L, M, P) \leftarrow$ UpdateReduce(L, M, P, h);
return $G = \{f \in M : \mathsf{LM}(f) \in L\}$;

Algorithm 3: Algorithm UpdateReduce

Input: $L \subset \mathcal{M}, M \subset \mathcal{R}, P \subset \mathcal{M} \times \mathcal{M}, f \in \langle M \rangle$, where (L, M) satisfies the M4GB-invariant (Def. 4.1) and f is full-reduced by G.
Output: Updated $(\hat{L}, \hat{M}, \hat{P})$, such that $f \in \hat{M}$, $\mathsf{LM}(f) \in \hat{L}$ and $(\hat{L}, \hat{M})$ satisfies the M4GB-invariant.

// *Compute tail-reduced set* H *to tail-reduce* $M \cup H$ *by* $G \cup \{f\}$
$H \leftarrow \{(\mathsf{LC}(f))^{-1} \cdot f\}$;
while $\exists u \in (\mathsf{Mono}(\mathsf{Tail}(M \cup H)) \setminus \mathsf{LM}(H)) : \mathsf{LM}(f) \mid u$ **do**
  // *LM of largest multiple of* f *we still need to insert to* H
  $u \leftarrow \max\{u \in (\mathsf{Mono}(\mathsf{Tail}(M \cup H)) \setminus \mathsf{LM}(H)) : \mathsf{LM}(f) \mid u\}$;
  $(M, h) \leftarrow$ MulFullReduce$(L, M, u/\mathsf{LT}(f), \mathsf{Tail}(f))$;
  // $u + h$ *is tail-reduced (by* G*) multiple of* f *with LT* u
  $H \leftarrow H \cup \{u + h\}$;
// *Cancel all terms in tails in* $M \cup H$ *that are multiples of* $\mathsf{LM}(f)$
while $H \neq \emptyset$ **do**
  Take $h \in H$ with $\mathsf{LM}(h) = \min \mathsf{LM}(H)$;
  $H \leftarrow H \setminus \{h\}$;
  $H \leftarrow \{g - ch : g \in H, c$ coefficient of $\mathsf{LM}(h)$ in $\mathsf{Tail}(g)\}$;
  $M \leftarrow \{g - ch : g \in M, c$ coefficient of $\mathsf{LM}(h)$ in $\mathsf{Tail}(g)\}$;
  $M \leftarrow M \cup \{h\}$;
// *Call Update function over leading monomials*
$(\hat{L}, \hat{P}) \leftarrow$ Update$(L, P, \mathsf{LM}(f))$;
return $\hat{L}, \hat{M}, \hat{P}$;

Algorithm 4: Algorithm MulFullReduce

Input: $L \subset \mathcal{M}, M \subset \mathcal{R}, t \in \mathcal{T}, f \in \langle M \rangle$, where (L, M) satisfies the M4GB-invariant.
Output: $(\hat{M}, h)$ such that $h =$ FullReduce$(t \cdot f, G)$, $(L, \hat{M})$ satisfies the M4GB-invariant and $M \subseteq \hat{M}$.

$h \leftarrow 0$;
for $s \in \mathsf{Term}(f)$ **do**
  $r \leftarrow t \cdot s$;
  if r *reducible by* L **then**
    $(M, g) \leftarrow$ GetReductor(L, M, r);
    $h \leftarrow h - (r/\mathsf{LT}(g)) \cdot \mathsf{Tail}(g)$;
  else
    // r *is non-reducible*
    $h \leftarrow h + r$;
return (M, h);

Algorithm 5: Algorithm GetReductor

Input: $L \subset \mathcal{M}, M \subset \mathcal{R}, r \in \mathcal{T}$, where (L, M) satisfies the M4GB-invariant and r is reducible by L.
Output: $(\hat{M}, h)$ such that h is a tail-reduced multiple of some $f \in G$ with $\mathsf{LM}(h) = \mathsf{LM}(r)$, $(L, \hat{M})$ satisfies the M4GB-invariant and $M \subseteq \hat{M}$.

if $\mathsf{LM}(r) \in \mathsf{LM}(M)$ **then**
  return $(M, M[\mathsf{LM}(r)])$; // *Desired reductor present in* M
$f \leftarrow M[\mathsf{ReduceSel}(L, r)]$; // $f \in M$ *that can reduce* r
// *Reductor has LM* r *and full-reduced Tail* h
$(M, h) \leftarrow$ MulFullReduce$(L, M, r/\mathsf{LT}(f), \mathsf{Tail}(f))$;
return $(M \cup \{r + h\}, r + h)$;

basis G, but is meant to also contain tail-reduced multiples of the basis that have been used to reduce terms in intermediate computations. Due to Equation 1, we can directly use the (Gebauer-Möller) Update function on L instead of G. By the M4GB-invariant, every $f \in M$ has a unique leading monomial, thus for $u \in \mathsf{LM}(M)$ we will denote by $M[u]$ the unique $f \in M$ with $\mathsf{LM}(f) = u$. We use a new Update-function UpdateReduce which performs the Update, but also modifies M to be tail-reduced by the new basis G to satisfy the M4GB-invariant again.

Second, all intermediate computations are done on polynomials that are tail-reduced by G, often only on the full-reduced tail part. As seen in MulFullReduce (algorithm 4) that multiplies a polynomial with a term and full-reduces it, the output is directly computed as a sum over non-reducible terms and full-reduced polynomials. Specifically, instead of adding a reducible term r to the output to later cancel it again, it first obtains a tail-reduced basis multiple g with $\mathsf{LM}(g) = \mathsf{LM}(r)$ by calling GetReductor (algorithm 5). It then subtracts the full-reduced tail $\mathsf{Tail}(g)$ of g multiplied with the correct scalar $r/\mathsf{LT}(g)$. In turn, GetReductor either simply returns the desired reductor if it's present in M, or computes it by calling MulFullReduce and includes it in M. Note that GetReductor(L, M, r) calls MulFullReduce(L, M, t, f) with $\mathsf{LM}(t \cdot f) \prec r$, which in turn calls GetReductor(L, M, u) only for

$u \leq \mathsf{LM}(t \cdot f)$. Thus the sequence $r_1, \ldots$ of monomials r input to GetReductor in any recursion is monotonic decreasing and as $\prec$ is a well ordering any recursion is finite.

4.1 Performance

In algorithm 2 we have presented a slightly simplified version for ease of exposition. For any efficient implementation of M4GB one should consider the following necessary improvements:

In our description, UpdateReduce proactively tail-reduces the entire M by any new basis element f. It is much more efficient to implement this in a lazy manner, e.g., as follows. By numbering each call to UpdateReduce, one tracks for every element of M between which two UpdateReduce calls it was last tail-reduced. The function GetReductor is modified in step 2 such that before returning an element of M, it first updates it by tail-reducing with the current basis.

Operations on tails of $f \in M$ are scalar multiplications and additions which are more efficient to compute if these tails are stored as truncated (or sparse) coefficient vectors relative to a global list of non-reducible monomials. This will also benefit the implementation of step 2 and 3 of UpdateReduce.

M4GB performs computations on leading terms and tails separately and frequently retrieves polynomials by their leading monomial. The most efficient data representation of M is a hash-map mapping the leading monomial to the leading coefficient and tail.

The computation of a critical pair half is identical to computation of a reductor. This allows two improvements for a critical pair (f, g) with LCM u. First, if for all $h \in M$ one remembers $\mathsf{LM}(\hat{f})$ of the $\hat{f}$ it was computed from then one can detect whether the critical pair half of f or g is identical to h, namely when $\mathsf{LM}(h) = u$ and $\mathsf{LM}(\hat{f})$ equals f_{LM} or g_{LM}, respectively. Secondly, if no $h \in M$ with $\mathsf{LM}(h) = u$ exists, one can insert either critical pair half in M (with leading monomial u) to avoid this being computed again.

4.2 Comparison with F4

M4GB has improvements to Buchberger's algorithm that are similar to those of F4. Namely both use efficient linear algebra to full-reduce S-polynomials. F4 achieves this by translating the full reduction of many S-polynomials to the row reduction of a particular matrix, while M4GB performs addition and scalar multiplication of tails stored as coefficient vectors.

Furthermore, both keep track of prior reduced elements to speed up computations. In F4 this is achieved by keeping all row reduced matrices and uses a function Simplify in building the next matrix that can reuse reduced rows of the previous matrices. In M4GB, this is clearly achieved by the set M that not only contains the tail-reduced basis elements, but tail-reduced multiples thereof as well that have been used as reductors.

It should be noted that M4GB has a more native description instead of translating desired computations into an external linear algebra computation. However, there are even more important differences between M4GB and F4 that would indicate M4GB is more computational and memory efficient.

Firstly, M4GB operates on single critical pairs in contrast to F4 that selects many critical pairs. In fact for F4, the best known selection of critical pairs is to select all critical pairs with the same lowest degree of the LCM. This seems to be the most efficient even if more critical pairs are selected than necessary, because it reduces the overhead associated in translating to and from matrices. Also, reducing the number of critical pairs may not significantly decrease matrix size anyway. Unfortunately, when a relatively small number of these critical pairs would be sufficient, F4 wastes a significant amount of computation and memory. This is easily seen in the complexity graph for a growing set of problems where there are significant 'jumps' whenever the degree of regularity (the largest LCM degree of critical pairs that was processed) increases. By operating on single critical pairs, M4GB does not have this disadvantage as can be seen from our experiments.

(a) F4 matrix **(b) M4GB's M**

Columns represent monomials, with non-reducible monomials last. Rows represent polynomials, with S-polynomial halves at the bottom (CD) and reductor polynomials at the top (AB).

Figure 1: Visualization of F4's matrix and M4GB's M

Secondly, M4GB operates mostly on coefficient vectors over non-reducible monomials, which in effect eliminates unnecessary computations and memory use involving reducible monomials. In Figure 1 we have visualized M4GB's M against matrices used by F4 to showcase this benefit. One can directly see that F4 works with a upper-triangular matrix A as well as larger matrices C and D. Note that for M4GB's M, instead of representing A as a diagonal matrix, we represent A as a single column of leading terms (instead of coefficients), or equivalently as the labels for the coefficient rows in B. It is well known that matrices generated by F4 have special structure and most pivots for the row reduction are known beforehand, namely those in A related to the reductor polynomials. Linear algebra software can take advantage of this special structure, but inherently must spent computation and memory in keeping track of coefficients related to reducible monomials. It contrast, whenever a reducible term arises, M4GB directly acts on this and reduces it without ever having to store the reducible term in the result.

5 IMPLEMENTATION

5.1 M4GB for dense over-defined systems

As a proof-of-concept we have implemented our Gröbner basis algorithm M4GB for a specific type of inputs, namely for dense overdefined systems over small finite fields. These type of systems have at most one solution with overwhelming probability. In particular we will assess our implementation on the subset of MQ-problems, which are one of few options for the next generation of cryptographic public key systems that are resistant against quantum attacks, and for which Yasuda et al. [16] started public challenges. We like to stress that our design choices for this implementation are not inherent to M4GB and we aim to also provide

more variations that are more suitable for more generic (sparse) systems in future work.

We are strong supporters of open-source implementations for numerous reasons for the benefit of the research community including the public verifiability of our results, availability and enabling further research and improvements by the community. Our source-code, written in C++11, is available under the GPLv3 license at:

`https://github.com/cr-marcstevens/m4gb`.

In the remainder of this section we will explain some important design choices for this particular implementation of M4GB, followed by a performance comparison against some available closed-source and open-source software, and a description of our efforts to break some of the MQ-challenges.

5.2 Design choices

Here we list our main design choices, we will refer to our source code page for more details. Our implementation consists mainly of the following components: a library implementing basic components (finite field, monomials, polynomials, threadpool, parser), a templated 'solver' implementing the main algorithm, and a command-line front-end that instantiates the solver and handles input, output and runtime statistics. We also provide two other 'solvers' that are simple wrappers around the publicly available closed-source `FGBlib` library and the open-source `OpenF4` library to facilitate comparisons with these libraries. Our implementation is configured compile-time as it is intended to run on problems where the run-time is significantly larger than compile-time, it allows simpler code and it results in more efficient compiled code.

We limited our implementation to the typically most performant monomial ordering, namely the degree reverse lexicographical ordering. It is more convenient to store and compare machine-size integers than exponent vectors representing monomials, therefore most implementations perform order-preserving translations between monomials and integers. OpenF4 and some other known implementations use a look-up table listing all monomials up to some degree, however this uses a significant amount of memory. Instead, we implemented a fast order-preserving encoder/decoder for monomials that utilizes small look-up tables that fit CPU caches.

For cryptographic MQ-problems the underlying finite field $\mathbb{F}_q$ is typically small with $q \leq 256$. We have therefore chosen to implement finite field operations in a relatively simple way by using log and reverse log look-up tables, which remains performant at least up to $q \leq 65536$. For $q \leq 256$ we use a multiplication look-up table that easily fit CPU cache. For odd $q \leq 31$ we further use a 3-to-1 multiply-and-add look-up table, as vector multiply-and-add with a fixed scalar can use a 2-to-1 look-up subtable of this. It is well-known that SIMD CPU instructions can provide significant speed improvements to vector operations, we hope to provide further optimized implementations and wrappers around open-source finite field libraries in the near-future.

Tails of $g \in M$ are represented by coefficient vectors relative to a global list of non-reducible monomials in increasing order and are truncated by removing trailing zeros. As our implementation aims at very dense systems, this list is the exhaustive list of non-reducible monomials up to the largest LCM of critical pairs. This choice enables a simpler implementation that does not have to deal with ad-hoc column insertions, but also allows a faster way to determine all reducible and non-reducible monomials using an approach comparable to the sieve of Eratosthenes. Finally, we process critical pairs in small batches to facilitate multi-threading and to amortize the cost of column removals when previously non-reducible monomials become reducible.

Table 1: Benchmark for the $m = 2n$ testcases over $\mathbb{F}_{31}$.

		Total CPU time (sec)			
n	m	OpenF4	FGb	Magma (projected)	M4GB
20	40	206	470	232.17	57
21	42	472	1002	500.26	170
22	44	1145	3118	1616.73	424
23	46	2274	6849	3184.82	1060
24	48	10293	64700	31167.61	2556
25	50	-	151653	77678.58	5575
26	52	-	360055	183628.74	15517
27	54	-	767543	409451.87	46548
		Memory (MB)			
20	40	4240	112	361.84	73
21	42	6640	165	577.34	121
22	44	14368	525	853.84	226
23	46	26135	918	1324.16	395
24	48	161945	1561	8872.94	663
25	50	-	2765	19718.78	1471
26	52	-	4607	25197	3328
27	54	-	8180	39844.84	6799

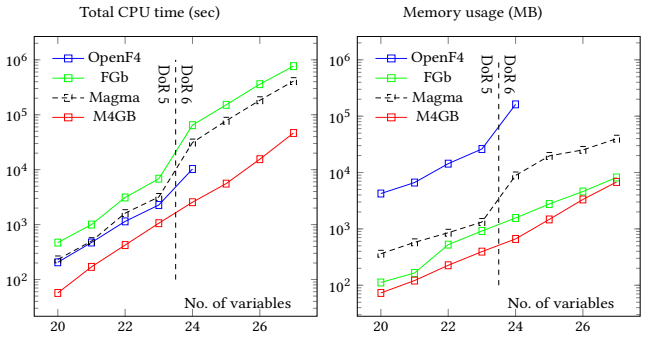

Figure 2: Results for $m = 2n$

5.3 Software performance comparison

We have compared our proof-of-concept implementation of M4GB against the following state-of-the-art Gröbner basis implementation: (1) FGBlib version 1.68 [9], (2) Magma version 2.20-6 [2], and (3) OpenF4 version 1.0.1 [5]. FGBlib and Magma are arguably the most efficient known implementations of the F4 and F5 algorithms, but are closed-source, and Magma even requires a paid license. OpenF4 is a recent open-source implementation of the F4 algorithm using SIMD CPU instructions, which makes it also an attractive competitor to other existing closed-source implementation.

Table 2: Benchmark for the $m = n + 1$ testcases over $\mathbb{F}_{31}$.

		Total CPU time (sec)			
n	m	OpenF4	FGb	Magma (projected)	M4GB
10	11	2.99	5	3.29	0.98
11	12	8.73	21	11.172	2.6
12	13	36.76	134	59.08	13.92
13	14	172.49	642	286.4	58.18
14	15	1258	5850	2810.75	393.19
15	16	7225	36361	17265.5	2424
		Memory (MB)			
10	11	101	33	32.09	17
11	12	341	50	64.12	16
12	13	1463	112	113.59	31
13	14	7622	323	281.53	74
14	15	33460	1098	1104	250
15	16	117396	4118	3320	837

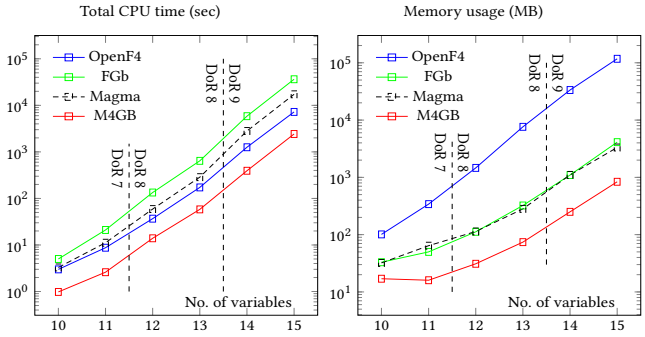

Figure 3: Results for $m = n + 1$

Two different machines have been used for benchmarking: (1) our dual Intel Xeon E5-2650v3 system with 128GiB RAM, (2) an external quad Intel Xeon E5-4640 system with 132GiB RAM. The first has been used to compare M4GB with FGBlib and OpenF4, while the second was needed to be able to compare M4GB with Magma since Magma requires a license. The run-times for Magma on this external machine have been projected to estimated run-times on our machine to allow comparisons in orders of magnitude in one graph. Despite this, please note that the margin of error should be negligible compared to the factor difference in orders of magnitude between the different software implementations. For the same reason, we have not run tests multiple times as is typically used to reduce margin of error. Also, both machines are non-uniform memory access (NUMA) machines, where memory is partitioned over the CPU chips. To avoid hidden costs related to process and memory transfers between CPU chips, we forced processes to a particular CPU chip and its associated memory using `numactl`. We could not disable Hyper-Threading, but ran processes only on physical cores.

We have benchmarked these implementations on MQ-problems similar to the MQ-challenges, specifically quadratic multivariate polynomial systems of m polynomial equations over n variables with randomly selected coefficients. We have limited ourselves to $\mathbb{F}_{31}$, as $\mathbb{F}_2$ requires more specialized implementations and $\mathbb{F}_{256}$ is not supported by FGBlib. Since the MQ-challenges were selected to all take more than a month using Magma, we have generated our own random systems that are computable in more reasonable time in the same manner as the MQ-challenges. Let $\mathbb{F}$ be a field, n, m be positive integers, and $\deg(f)$ denotes the maximum degree of monomials of f. We define the following set

$$MQ(\mathbb{F}, n, m) = \{(f_1, \ldots, f_m) \in (\mathbb{F}[x_1, \ldots, x_n])^m : \deg(f_i) = 2\}.$$

We consider two type of systems similar to the MQ-challenge types III and VI. Firstly, strongly over-defined systems taken from the set $MQ(\mathbb{F}_{31}, n, 2n)$, for $n = 20, 21, \ldots, 27$ and the constant terms in the systems have been adjusted to match a known randomly-selected solution. Secondly, weakly over-defined systems taken from the set $MQ(\mathbb{F}_{31}, n, n + 1)$ with $n = 10, 11, \ldots, 15$.

Our benchmarking results for $m = 2n$ and $n = 20, \ldots, 27$ are listed in Table 1 and shown in Figure 2. Between FGBlib, Magma and OpenF4 we see a clear trade-off between CPU speed and memory usage, with OpenF4 being the fastest followed by Magma and then FGBlib, and FGBlib being the most memory efficient followed by Magma and then OpenF4. In particular, OpenF4 is faster than Magma by a factor 1.05 up to a factor 3, probably because of its SIMD implementation. However, OpenF4 uses the most memory of all by at least a factor 11 to Magma, which prevents us to run OpenF4 for $n \geq 25$.

We observe that M4GB is the fastest of all and is at least a factor 2.15 and up to a factor 4 faster than OpenF4 (respectively just before and after the increase in *degree of regularity* (DoR)). Moreover, M4GB also uses the least memory of all by a factor between 1.2 and 2.35 compared to the second-best FGBlib. However, we do note that somehow memory usage of our implementation seems to grow more strongly than FGBlib's and Magma's, which may be related to our particular choice to use coefficient vectors over all non-reducible monomials up to some bound.

One can also clearly observe M4GB's benefit of processing critical pairs in small batches, where the increase in degree of regularity does not cause a sudden factor increase in run-time.

Our benchmarking results for $m = n + 1$ and $n = 10, \ldots, 15$ are listed in Table 2 and shown in Figure 3. Again one can observe the same ordering in run-time (M4GB is fastest followed by OpenF4, Magma and FGBlib) and similarly for memory usage (M4GB uses the least, followed by comparable FGBlib and Magma, and lastly OpenF4). Here we observe that M4GB is faster by a factor 2.6 up to 3.3 compared to second-best OpenF4 and uses less memory by a factor 1.9 up to 4.4 compared to the second-best (Magma or FGBlib).

6 BREAKING MQ-CHALLENGES TYPE V & VI

In this part, we discuss our efforts in breaking several MQ-challenges of Type V and Type VI (under-defined systems over $\mathbb{F}_{256}$ and $\mathbb{F}_{31}$, respectively). The following machines were used to solve some parameters in MQ-challenge:

A) Desktop machine with Intel(R) Core(TM) i7-2600K CPU @ 3.40GHz and 16GB RAM
B) NUMA machine with two nodes of Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz processors and 128GB RAM each.

A summary of broken challenges using these machines in shown in Table 3.

Table 3: Summary on MQ-challenges we broke with M4GB.

Type	n/m	Machine Used	# Node	Duration
V	24/16	A	1	$\approx$ 9.3 hours
V	25/17	B	1	$\approx$ 46.33 hours
V	27/18	B	2	$\approx$ 10.9 days
VI	24/16	A	1	$\approx$ 1.2 hours
VI	25/17	B	1	$\approx$ 9.87 hours
VI	27/18	B	1	$\approx$ 31.48 hours
VI	28/19	B	2	$\approx$ 7.61 days

Both MQ-challenge types have more variables than equations ($n \approx 1.5m$) and they represent equation systems related to MQ-problem based cryptographic public key signature schemes. There will be a large number of solutions due to the small number of equations, however the Gröbner basis corresponding to this set of solutions will be very costly to compute. We therefore use the hybrid approach by Bettale et al. [1] which is a trade-off between exhaustive search and Gröbner bases computations. The main idea is to go over all possible values for k chosen variables and compute the Gröbner basis of the m equations over just $n-k$ variables. More specifically, we first guess $n-m$ variables to force a square system with equal number of equations and variables, which has probability e^{-1} to have a solution (e.g., see [11]). Then we go over all possible values over 1 or 2 remaining variables and try to solve the resulting weakly over-defined systems. If this fails then we select other values for the $n-m$ variables and retry.

This approach has the benefit that we can actually run multiple independent processes simultaneously (e.g., over different NUMA nodes or different machines). Moreover, we can also practically estimate the average time as well as worst-case time to obtain a solution for the whole system by computing a Gröbner basis for one of the subsystems. For instance for type VI with $m = 16$, solving one subsystem ($m = 16, n = 15$) took 1 hour 10 minutes wall-clock time using a single thread on machine A. An average total CPU-time estimate to solve type VI systems over $\mathbb{F}_{31}$ should then be approximately 18.7 hours. It uses 837MB of memory for each subsystem, which allowed the simultaneous computation on 8 subsystems well within the available 16GB of RAM. We broke both MQ-challenges of type V and VI for $m = 16, n = 15$ on this desktop machine in 9.3 hours and 1.2 hours wall-clock time respectively, significantly faster than the designed minimum cost of a month on a Xeon system using Magma.

For the larger MQ-challenges of type V and VI, we used machine B. We ran 10 simultaneous Gröbner bases computations using M4GB, each with 2 threads and forced to one NUMA node using the `numactl` program.

For type VI with $m = 19$ we slightly modify our strategy in computing Gröbner bases for all subsytems with $m = 19, n = 18$. An important observation here is that in the final stage, M4GB takes a significant amount of time using only a single thread. Thus, instead of waiting the computation to finish, one can start to compute Gröbner basis for the next subsystem as soon as this last stage of the previous computation begin. In this way all processors are fully occupied so that the time to obtain a solution can be significantly reduced. We used both NUMA nodes where one process was run in each node using 10 threads. A solution was found after running the computation for roughly 7.6 days.

7 FUTURE WORK

In future work we aim to implement variations that are more suitable for sparse polynomial systems, in particular by replacing the design choice to use the exhaustive list of non-reducible monomials up to some bound. Also, further speed-ups may be gained via SIMD CPU instructions or the use of GPUs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their feedback and suggestions. We also would like to thank Ronald van Luijk and Marco Streng for the use of their machine to benchmark Magma, including helping us preparing the machine according to our need. This machine and machine 'B)' were funded by The Netherlands Organisation for Scientific Research (NWO) via the "Vernieuwingsimpuls" career grants of respectively Ronald van Luijk and Marco Streng, and Marc Stevens.

REFERENCES

[1] Luk Bettale, Jean-Charles Faugère, and Ludovic Perret, *Hybrid Approach for Solving Multivariate Systems over Finite Fields*, J. Mathematical Cryptology **3** (2009), no. 3, 177–197.
[2] Wieb Bosma, John J. Cannon, and Catherine Playoust, *The Magma Algebra System I: The User Language*, J. Symb. Comput. **24** (1997), no. 3/4, 235–265.
[3] B. Buchberger, *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal*, Ph.D. thesis, University of Innsbruck, 1965.
[4] Johannes A. Buchmann, Jintai Ding, Mohamed Saied Emam Mohamed, and Wael Said Abd Elmageed Mohamed, *MutantXL: Solving Multivariate Polynomial Equations for Cryptanalysis*, Symmetric Cryptography (Dagstuhl, Germany) (Helena Handschuh, Stefan Lucks, Bart Preneel, and Phillip Rogaway, eds.), Dagstuhl Seminar Proceedings, no. 09031, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.
[5] Titouan Coladon, Vanessa Vitse, and Antoine Joux, *OpenF4 implementation*, https://github.com/nauotit/openf4, 2015.
[6] Nicolas Courtois, Alexander Klimov, Jacques Patarin, and Adi Shamir, *Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations*, Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding (Bart Preneel, ed.), Lecture Notes in Computer Science, vol. 1807, Springer, 2000, pp. 392–407.
[7] *The Certicom ECC Challenge*, https://www.certicom.com/index.php/the-certicom-ecc-challenge, 1997.
[8] Jean-Charles Faugère, *A New Efficient Algorithm for Computing Gröbner Bases (F4)*, Journal of Pure and Applied Algebra **139** (1999), no. 1–3, 61–88.
[9] Jean-Charles Faugère, *FGb: A Library for Computing Gröbner Bases*, Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings (Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, eds.), Lecture Notes in Computer Science, vol. 6327, Springer, 2010, pp. 84–87.
[10] Jean-Charles Faugère, Patrizia M. Gianni, Daniel Lazard, and Teo Mora, *Efficient Computation of Zero-Dimensional Gröbner Bases by Change of Ordering*, J. Symb. Comput. **16** (1993), no. 4, 329–344.
[11] Giordano Fusco and Eric Bach, *Phase Transition of Multivariate Polynomial Systems*, Mathematical Structures in Computer Science **19** (2009), 9–23.
[12] Rüdiger Gebauer and H. Michael Möller, *On an Installation of Buchberger's Algorithm*, J. Symb. Comput. **6** (1988), no. 2/3, 275–286.
[13] *The RSA Factoring Challenge*, http://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge.htm, 1991.
[14] Peter W. Shor, *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, SIAM Review **41** (1999), no. 2, 303–332.
[15] *The Lattice Challenge*, http://www.latticechallenge.org/, 2010.
[16] Takanori Yasuda, Xavier Dahan, Yun-Ju Huang, Tsuyoshi Takagi, and Kouichi Sakurai, *MQ Challenge: Hardness Evaluation of Solving Multivariate Quadratic Problems*, IACR Cryptology ePrint Archive **2015** (2015), 275.